第 7 章

「適当に構文解析する話 (575)」by Hinata (72)

7.1 はじめに

おはようございます。72 回生の Hinata と申します。今年ついに高校生となり、老いを実感する日々を送っております。時の流れは速いですね……。2 年半くらいプログラミングを続けてきましたが、ここ 2 年弱くらいは Twitter や Twitter や Twitter に張り付いてばかりで全く精進をしていないため、一向に成長する様子がなく悲しい気分になっています。ですので、今年こそはちゃんと精進して†圧倒的成長†を掴み取る所存でございます。

今回のタイトルは「適当に構文解析する話 (575)」です。こんなことを言うと「構文解析は 8 音だから 575 ではなく 585 だ!」とおっしゃる方もいるかもしれません。その通りです。本当に申し訳ございません。許してヒヤシンス (・ω<)

さて、皆さんは構文解析を実装したことがありますか? 構文解析というレベルじゃなくても、大抵の人は文字列をパースしたことならあるでしょう。ない人はごめんなさい。その時、どのように実装しましたか? 正規表現を使った、という人が多いと思います。使ってない人はごめんなさい。

確かに、正規表現は強力です。単純な文字列なら簡単にパースできます。例えば、2017/05/02 のような日付にマッチする正規表現は `/\d{4}\/\d{2}\/\d{2}/` となります。正規表現もそれなりに簡潔でわかりやすいですね。数字 4 文字、スラッシュ、数字 2 文字、スラッシュ、数字 2 文字という文字列、つまり日付にマッチします。

確かに正規表現は強力ですが、弱点もあります。可読性です。そもそも正規表現は言語としてみれば比較的読みにくい部類に入ると思います。可読性が低いというのは即ち保守性が低いということです。仕様を変更するたびに長々とした正規表現を編集しないといけないなんて、考えただけでぞっとします。正規表現は簡潔なものじゃないと許されません。

では、正規表現以外にどのような方法があるでしょうか? 簡単です。文字列をパースするパーサを作ればいいのです。この記事では JSON パーサを実装します。もちろん、正規表現だけでやろうなんて難易度ルナティックなことはしません。我々は賢いので。

具体的なパーサの実装方法にもいろいろあります。言語によっては文法を表した文字列 (BNF 記法とか) からパーサを生成するパーサジェネレータがライブラリとして提供されている場合があります。BNF 記法で文法を表現してパーサジェネレータにぶち込むだけで楽なのですが、それだと仕組

みがよく分からないので、今回はパーサコンビネータというものを実装しようと思います。

7.2 がいよう

パーサコンビネータとはなんでしょう? 当然、パーサを組み合わせるもの (コンビネータ) です。

パーサは入力を引数に取りパースした結果を返す関数、パーサコンビネータはパーサを引数に取り新たなパーサを返す高階関数とみなせます。高階関数とか言ってる時点で分かる通り、一般に関数型言語と呼ばれる言語で扱うといい感じらしいです。パーサコンビネータを調べると Haskell とか Scala とかそのあたりの言語の話がたくさん出てきますし。ちなみにこの記事で使う言語は JavaScript(JS) です。JS は関数型言語とはお世辞にも言い難いですが、関数が第一級オブジェクトですから高階関数も実装できます。問題なんか何もないよ♪

パーサの入力はパースする文字列とパースを始める位置。返り値はパースが成功したどうか、パース結果、そして新しい読み取り位置です。ね、簡単でしょう?

7.3 じっそう

簡単なわけないだろいい加減にしろ

というわけで、実装しながら理解していきましょう。

例えば、hoge というトークン (字句) をパースするパーサは次のようになります。

List 7.1 parseHoge

```
function parseHoge(str, position) {
    if (str.substr(position, 4) === "hoge") {
        return {
            success: true,
            match: "hoge",
            position: position + 4
        };
    } else {
        return {
            success: false,
            position // position: position の糖衣構文(syntax sugar)
        };
    }
}

// アホ毛っていいよね
parseHoge("ahoge", 1); // { success: true, match: "hoge", position: 5 }
parseHoge("ahoka", 1); // { success: false, position: 1 }
```

しかし、これでは汎用性が低すぎます。アホ毛を検出するくらいしかできません。任意の文字列をパースできるようにしてみましょう。

List 7.2 string 関数

```
function string(str) {
    const length = str.length;

    return function(target, position) {
        if (target.substr(position, length) === str) {
            return {
                success: true,
                match: str,
                position: position + length
            };
        } else {
            return {
                success: false,
                position
```

```
            };
        }
    }
}

// fugaってイタリア語でフーガって意味なんですよ！
const fuga = string("fuga");
fuga("fuga", 0); // { success: true, match: "fuga", position: 4 }
// カノン(音楽用語)はドイツ語でkanonです。他意はありません。
fuga("Kanon", 0); // { success: false, position: 0 }
```

とりあえずパーサができたので、パーサを引数に取り新しいパーサを返すパーサコンビネータを試しに実装してみます。

repeat 関数は繰り返しを表現するパーサを返します。

List 7.3 repeat 関数

```
function repeat(parser) {
    return function(str, position) {
        let currentPosition = position;
        const matchResults = [];
        while (true) {
            const parseResult = parser(str, currentPosition);
            if (parseResult.success) {
                matchResults.push(parseResult.match);
                currentPosition = parseResult.position;
            } else {
                break;
            }
        }
        return {
            success: true,
            match: matchResults,
            position: currentPosition
        };
    };
}

const hogehoge = repeat(string("hoge")); // /(hoge)*/ という正規表現と同じ働き
hogehoge("ahogehoge", 1); // { success: true, match: ["hoge", "hoge"], position: 9 }
hogehoge("ahoka", 1); // { success: true, match: [], position: 1 }
```

次に、パーサを連結する seq 関数と、パーサを複数受け取って順次適用していく alt 関数を作ります。

List 7.4 seq 関数

```
function seq(...parsers) {
    return function(str, position) {
        const matchResults = [];
        let currentPosition = position;
        for (const parser of parsers) {
            const result = parser(str, currentPosition);
            if (!result.success) {
                // パーサが1つでも失敗したら全体として失敗する
                return {
                    success: false,
                    position
                }
            }
            matchResults.push(result.match);
            currentPosition = result.position;
        }
        return {
            success: true,
            match: matchResults,
            position: currentPosition
        };
    };
}
const hogefuga = seq(string("hoge"), string("fuga"));
hogefuga("hogefuga", 0); // { success: true, match: ["hoge", "fuga"], position: 8 }

// hogeにはマッチするが、fugaにはマッチしないのでおしまい
hogefuga("hogepiyo", 0); // { success: false, position: 0 }
```

List 7.5 alt 関数

```
function alt(...parsers) {
    return function(str, position) {
        for (const parser of parsers) {
            const result = parser(str, position);
            if (result.success) {
                return result;
            }
        }
        return {
            success: false,
            position
        };
    }
}

// /(hoge)|(fuga)/という正規表現と同じ働き
const hogeOrFuga = alt(string("hoge"), string("fuga"));
hogeOrFuga("hoge", 0); // { success: true, match: "hoge", position: 4 }
hogeOrFuga("fuga", 0); // { success: true, match: "fuga", position: 4 }
hogeOrFuga("piyo", 0); // { success: false, position: 0 }
```

オプションを表現する option 関数です。

List 7.6 option 関数

```
function option(parser) {
    return function(str, position) {
        const result = parser(str, position);
        if (result.success) {
            return result;
        } else {
            return {
                success: true,
                match: null,
                position
            }
        }
    };
}
```

いろいろ実装しましたが、単純な正規表現で要求を満たせるならそれで十分ですね。何も車輪の再発明をする必要はありません。そこで、正規表現にマッチすれば成功し、しなければ失敗するパーサを返す regexp 関数を実装します。1 つ目の引数は正規表現です。2 つ目の引数、group が指定された場合は正規表現のマッチ結果のうち、指定されたキャプチャグループの部分のみが結果として返ります。

List 7.7 regexp 関数

```
function regexp(re, group = 0) {
    if (re.multiline || re.source.startsWith("^")) {
        throw new Error("mフラグが設定されていない正規表現で^を使わないでください");
    }
    let flag = "y";
    if (re.multiline) { flag += "m"; }
    if (re.ignoreCase) { flag += "i"; }
    if (re.unicode) { flag += "u"; }
    const newRegexp = new RegExp(re.source, flag);
    return function(str, position) {
        newRegexp.lastIndex = position;
        const regexpResult = newRegexp.exec(str);
        if (regexpResult) {
            return {
                success: true,
                match: regexpResult[group],
                position: position + regexpResult[group].length
            };
        } else {
            return {
                success: false,
                position
            };
        }
```

```
    }
}

const digits = regexp(/\d+/); // 数字の列にマッチする正規表現です
digits("114514", 0); // { success: true, match: '114514', position: 6 }
digits("01234s", 0); // { success: true, match: '01234', position: 5 }
digits("NPCA", 0); // { success: false, position: 0 }
```

このコードでは sticky フラグを有効に活用しています。何をやっているのか分からない人も多数いるかとは思いますが、説明が面倒なので省かせてください。JavaScript の知識が必要になるので。

理想を言えば、false という文字列をパースすると false という値が返ってきてほしいところです。そこで、パース結果を加工する map 関数を実装します。

List 7.8 map 関数

```
function map(parser, fn) {
    return function(str, position) {
        const result = parser(str, position);
        if (result.success) {
            return {
                success: true,
                match: fn(result.match),
                position: result.position
            };
        } else {
            return {
                success: false,
                position
            };
        }
    };
}

// () => false は function() { return false; } と等価(アロー関数)
const falseLiteral = map(string("false"), () => false);
falseLiteral("false", 0); // { success: true, match: false, position: 5 }
falseLiteral("true", 0); // { success: false, position: 0 }
```

`match: false` だとマッチしていないかのようですが、パース結果が false というだけです。

さて、結構な数のパーサ、パーサコンビネータが揃いました。もう十分でしょう。そう思った人もいるとは思いますが、残念でした。

例えば、((())) のように括弧が何重にも入れ子になった文字列をパースすることを考えます。括弧の対応が取れていなければ失敗するようにしたいですね。

一見すると次のようにすれば良さそうです。

List 7.9 失敗例

```
const parenthesis = seq(string("("), option(parenthesis), string(")"));
```

しかし、これを実行してみると

ReferenceError: parenthesis is not defined

というエラーが出ます。option 関数の引数にある parenthesis という変数を参照しようとしますが、この時点で parenthesis という変数は存在しません。このように、変数が登場したらすぐ評価しようとする戦略のことを先行評価といいます。先行評価の対義語は遅延評価です。JS の言語仕様に遅延評価は存在しませんが、それっぽいことならできます。

List 7.10 lazy 関数

```
function lazy(callback) {
    let parser;
    return function(str, position) {
```

```
        if (!parser) {
            parser = callback();
        }
        return parser(str, position);
    };
}

const parenthesis = seq(
    string("("),
    option(lazy(() => parenthesis)),
    string(")")
);
parenthesis("(())", 0);
// { success: true, match: ["(", ["(", null, ")"], ")"], position: 4 }
```

lazy 関数は、パーサを返す関数を引数に取ります。引数の関数が実行されるのは最初にパースする 1 回のみです。これで、再帰的な文法もパースできますね。

最後に、文字列の終了を表す eof(end of file) パーサを実装します。正規表現における$です。実装といっても数行ですみますが。

List 7.11 eof パーサ

```
function eof(str, position) {
    if (str.length > position) {
        return {
            success: false,
            position
        };
    } else {
        return {
            success: true,
            match: null,
            position
        };
    }
}
```

7.4 くみたて

JSON パーサを作るための部品が揃ってきました。今まで実装してきたパーサやパーサコンビネータを組み合わせ、JSON パーサを実装しましょう。JSON の仕様書を確認しながら、文法の定義をそのままコードに起こせば簡単にパーサが作れます。

JSON の文法がわからない人は JSON の紹介を参考にしてください。文法は単純ですし、JS を知っている人ならすぐに理解できると思います。

まず、空白のパーサを作ります。JSON で許されている空白はスペース、タブ、CR、LF のみです。*は 0 回以上の繰り返しを表現するので、仮に空白がなくても空白パーサは失敗しません。

List 7.12 空白のパーサ

```
const whitespace = regexp(/[\x20\x09\x0A\x0D]*/);
```

次に、波括弧 (ブレース) や角形括弧 (ブラケット)、コロンやコンマ用のパーサを作ります。これらの文字の前後に空白が入ることがあるので注意しましょう。

List 7.13 記号類のパーサ

```
function tokenWithWhiteSpace(str) {
    return map(seq(whitespace, string(str), whitespace), result => result[1]);
}

const lbrace = tokenWithWhiteSpace("{");
const rbrace = tokenWithWhiteSpace("}");
```

```
const lbracket = tokenWithWhiteSpace("[");
const rbracket = tokenWithWhiteSpace("]");
const comma = tokenWithWhiteSpace(",");
const colon = tokenWithWhiteSpace(":");
```

seq 関数を使うと、余計な結果 (前後に挟まっている空白パーサの結果) も返ってきてしまうので、map 関数で真ん中の文字列パーサの結果のみを取り出しています。

次に、基本的な構成要素である `true` と `false`、そして `null` のパーサを作ります。

List 7.14 true、false、null のパーサ

```
const trueLiteral = map(string("true"), () => true);
const falseLiteral = map(string("false"), () => false);
const nullLiteral = map(string("null"), () => null);
```

次に、数値のパーサを作ります。0 や負の数、小数や指数表記に対応する必要がありますが、幸いなことに比較的簡潔な正規表現を使うだけですみます。

List 7.15 数値のパーサ

```
const numberLiteral = map(regexp(/-?(0|[1-9]\d*)(\.\d+)?([Ee][+-]?\d+)?/), Number);

numberLiteral("-3.34E1", 0); // { success: true, match: -33.4, position: 7 }
```

さっきは簡潔と言いましたが、実際に書いてみるとそんなに簡潔ではありませんでした。ちょっと分解してみましょう。

`-?` : マイナスの記号。オプション (あってもなくてもいい)。

`(0|[1-9]\d*)` : 0、または 0 以外の数字から始まる整数。つまり 0 以上の整数。必須。

`(\.\d+)?` : 小数点と小数部分。オプション。

`([Ee][+-]?\d+)?` : 指数部分。オプション。

前言と矛盾しているようですが、そんなに複雑ではないですね。

また、`Number` 関数は文字列を数値に変換する役割を果たしています。

さて、次は文字列のパーサです。

List 7.16 文字列のパーサ

```
const stringRegexp =  /"((\\(u[0-9A-Fa-f]{4}|["\\/bfnrt])|[^\\"\b\f\n\r\t])*)"/;
const escapeSequences = new Map([
    ["b", "\b"],
    ["f", "\f"],
    ["n", "\n"],
    ["r", "\r"],
    ["t", "\t"]
]);
function unescapeString(str) {
    return str.replace(/\\(u[0-9A-Fa-f]{4}|[^u])/g, (_, e) => {
        if (e[0] === "u") {
            return String.fromCharCode(parseInt(e.substr(1), 16));
        } else {
            if (escapeSequences.has(e[0])) {
                return escapeSequences.get(e[0]);
            } else {
                // バックスラッシュかスラッシュ
                return e[0];
            }
        }
    });
}
const stringLiteral = map(regexp(stringRegexp, 1), unescapeString);
```

正規表現で取り出した文字列を unescapeString 関数でアンエスケープしています。`\n` という文字列が改行に変換されたり、`\u3042` が「あ」に変換されたりします。

1 行目の正規表現がよく分からないという人もいるかと思いますが、正規表現を文章で説明するのって結構面倒なんですよ。さっき数値のパーサのところで 1 度やりましたが、もうやりたくありません。その代わり、正規表現が一発で理解できる素晴らしい図を用意しました。かなりわかりやすくなったのではないでしょうか。One of というのはいずれか、None of はいずれでもない、という意味です。

ちなみに、以下の画像は regexper で生成しました。このサイトは JS の正規表現を読み解く上で非常に便利なサイトですので、ぜひ使ってみてください。

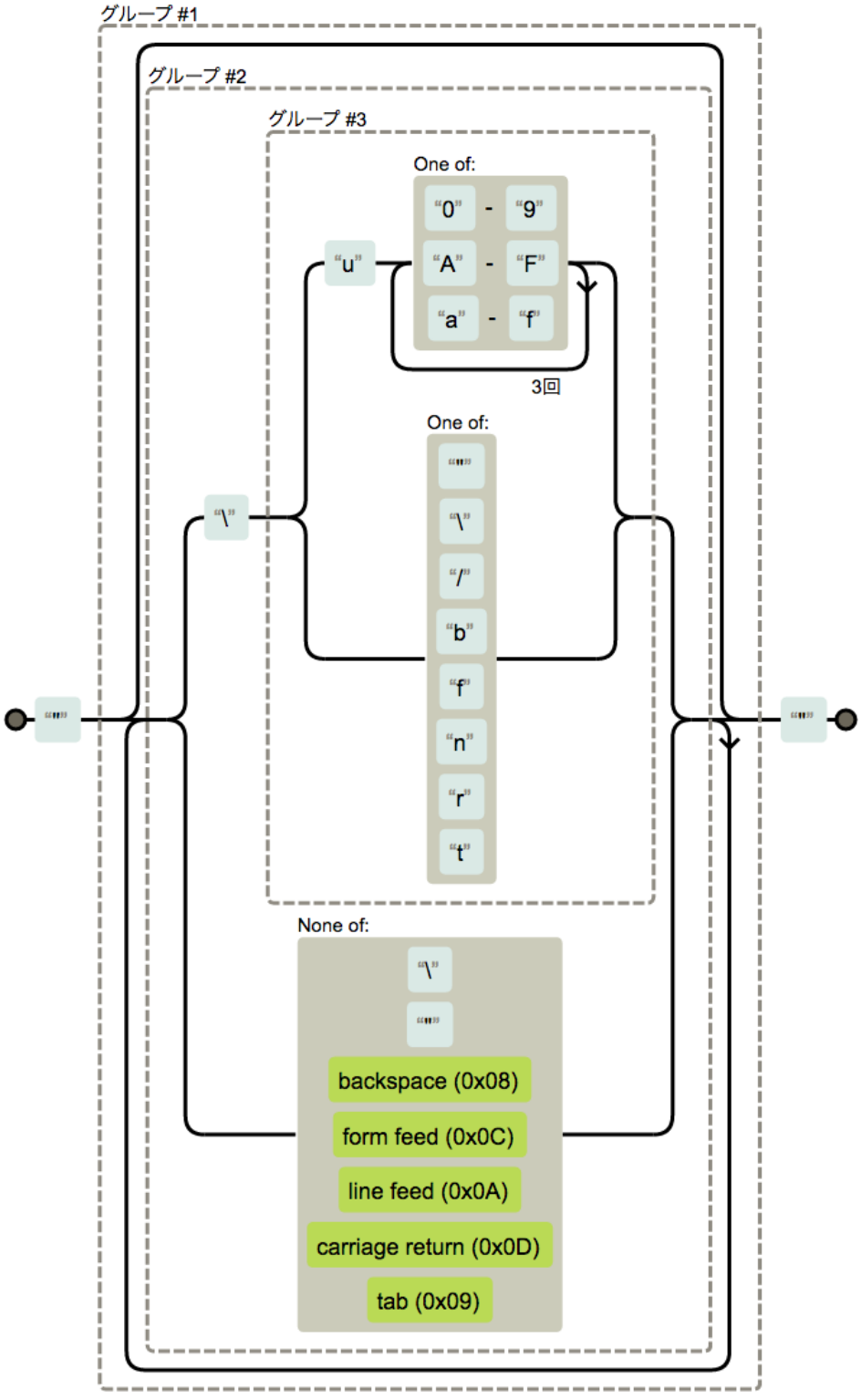
グループ #1
グループ #2
グループ #3
One of:
"0" - "9"
"A" - "F"
"a" - "f"
"u"
3回
One of:
""
"\"
"/"
"b"
"f"
"n"
"r"
"t"
"\"
""
None of:
"\"
""
backspace (0x08)
form feed (0x0C)
line feed (0x0A)
carriage return (0x0D)
tab (0x09)

図 7.1 正規表現の図

やはり図の力は偉大ですね。見れば分かると思うので特に説明はしません。

次に配列のパーサを実装したいのですが、そのためには値のパーサが必要になり、値のパーサを実装するには配列のパーサが必要になる、という循環参照が生まれてしまいます。そこで、値のパーサを lazy 関数を使って先に作ってしまいましょう。ちなみに、値というのは true、false、null、文字列、数値、オブジェクト、配列のうち 1 つです。

List 7.17 value パーサ

```
const value = lazy(() => alt(
    object,
    array,
    stringLiteral,
    numberLiteral,
    trueLiteral,
    falseLiteral,
    nullLiteral
));
```

あとは object と array を実装するだけです。配列は値をコンマ区切りで記述し、オブジェクトはキーと値のペアをコンマ区切りで記述します。共にコンマ区切りなので、コンマ区切りパーサを実装しましょう。

List 7.18 コンマ区切り表記のパーサ

```
function commaSeparated(parser) {
    const parserWithComma = map(seq(comma, parser), result => result[1]);
    return map(option(seq(parser, repeat(parserWithComma))), result => {
        if (result === null) {
            // 空文字列
            return [];
        }
        return [result[0]].concat(result[1]);
    });
}
const commaSeparatedNumber = commaSeparated(numberLiteral);
commaSeparatedNumber("", 0); // { success: true, match: [], position: 0 }
commaSeparatedNumber("2", 0); // { success: true, match: [2], position: 1 }
commaSeparatedNumber("2, 3, 4,", 0); // { success: true, match: [2, 3, 4], position: 7 }
```

最後に余分なコンマ (trailing comma と呼びます) が付いていますが、パースはその手前で止まっています。

あと少しで完成ですね。array と object のパーサを実装してしまいましょう。

List 7.19 array パーサ

```
const array = map(seq(lbracket, commaSeparated(value), rbracket), result => result[1]);
```

List 7.20 object パーサ

```
const keyValue = map(seq(stringLiteral, colon, value), result => [result[0], result[2]]);
const object = map(seq(lbrace, commaSeparated(keyValue), rbrace), result => {
    const plainObject = {};
    for (const pair of result[1]) {
        const key = pair[0];
        const value = pair[1];
        plainObject[key] = value;
    }
    return plainObject;
});
```

最後に、parseJSON 関数を実装しておしまいです。

List 7.21 parseJSON 関数

```
function parseJSON(jsonString) {
    const json = seq(value, eof);
```

```
    const parseResult = json(jsonString, 0);
    if (parseResult.success) {
        return parseResult.match[0];
    } else {
        throw new Error("Syntax Error");
    }
}
```

できた parseJSON 関数をテストしてみましょう。JS では JSON.stringify を使うことでオブジェクトを JSON 文字列に変換することができます。

List 7.22 parseJSON のテスト

```
parseJSON(JSON.stringify({
    title: "ゆゆ式",
    characters: [
        { name: "野々原ゆずこ", cv: "大久保瑠美" },
        { name: "櫟井唯", cv: "津田美波" },
        { name: "日向縁", cv: "種田梨沙" },
        { name: "相川千穂", cv: "茅野愛衣" },
        { name: "岡野佳", cv: "潘めぐみ", },
        { name: "長谷川ふみ", cv: "清水茉菜" },
        { name: "松本頼子", cv: "堀江由衣" }
    ]
})); // 成功する

parseJSON("[1, 2, 3"); // Syntax Error
```

無事 JSON パーサができました。成し遂げたぜ。

7.5 おわりに

最後まで読んでいただき、ありがとうございました。構文解析という慣れないテーマを題材にしたことを考えると、かなりしっかり書けたと思っています。

思い出してみれば、部誌のテーマを何にしようかと悩んでいる頃、参考文献の 1 つ目のサイトにたどり着き、パーサコンビネータを知ったのがきっかけでした。じゃあこれ部誌に書いたらいいじゃんということで書きはじめたのですが、今年からはカタログというものができ、その締め切りが 4 月 10 日でした。その頃は JSON パーサの J の字もなかったのに、何を思ったか、勢いに任せてカタログで「JSON パーサを作ります」と宣言してしまったのです。実装の道筋だけは見えていたのですが、実装が非常に遅いことに定評のある私が期日を守れるのか心配でした。完成しないと予告詐欺になってしまいますからね。

イチから——いいえ、ゼロから始めた JSON パーサ実装計画ですが、無事完成してよかったです、本当に。文法が単純な JSON を選んだ過去の私を褒めたい気分です。ちなみにリゼロは見てません。

また来年の部誌でお会いしましょう。それではまた。

7.6 さんこうぶんけん

JavaScript でパーサコンビネータのコンセプトを理解する
parsimmon: A monadic LL(infinity) parser combinator library for javascript - GitHub
JSON の紹介 - json.org
RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format